The Relational Data Model: Functional-Dependency Theory

1 Overview

- This handout builds on the concept of functional dependencies and normal forms.
- Remember that the whole on-going idea is for us to build a good conceptual framework so that we can systematically reason about and test relational schemas.
- Although personally, I just like the idea of picking up any kind of framework that has a strong mathematical foundation.

2 Closure of Attribute Sets

- Here's a key question: given some attribute set α, can we find the set of all attributes B such that $\alpha \rightarrow B$ for a given set F of functional dependencies?
- This set is called the *closure of* α *under* F, written as α^+.
- Finding α^+ is useful for a number of reasons:
  - It will tell us if α is a superkey.
  - It can tell us if, given another attribute set β, $\alpha \rightarrow \beta$.
  - It is another way to find F^+ (an alternative to the algorithm from the last handout).
- Here's the algorithm:

```
result := alpha;
while (result is changing) do
    for each functional dependency (beta -> gamma) in F do {
        if beta is a subset of result {
            result := result union gamma;
        }
    }
```

- In less technical language, we start with α itself as the result set. The algorithm then goes through all of the functional dependencies in F. If the left side of the functional dependency is a subset of the result set, then we add/union the right side of the dependency to that result set. We do this for every functional dependency, possibly multiple times, until we encounter a pass where no more changes are made to the result set of attributes. At that point, we're done.
- The rationale for this algorithm comes from the inference rules that have been discussed before; specifically, proper applications of the reflexivity, transitivity, and union rules.

3 Canonical Cover

- Another theoretical result with practical implications is the determination of a *canonical cover*.
- The canonical cover F_c for a set of functional dependencies F can be viewed as a "smaller but equivalent" set of functional dependencies.
- Why do we care? One of the practical uses of functional dependencies is for checking the validity of a data update. When a change to the database is requested, we want to make sure that this change will not violate the set of functional dependencies that holds on the database. If the change may violate a functional dependency (e.g., introduce a duplicate key), then we can reject the change.
- This check is more efficient for a smaller set of functional dependencies. That's where canonical cover comes in — expresses the same constraints as the original set of functional dependencies, but is likely smaller in size.
- So, how do we express this idea more formally?
  - When are two sets of functional dependencies "equivalent," anyway? If you think about it, they are equivalent if their closures are the same.
  - We say that an attribute of a functional dependency is *extraneous* if removing the attribute does not change F^+. That is, given a set F of functional dependencies and some functional dependency $\alpha \rightarrow \beta \in F$:
    - An attribute A is extraneous in α if $A \in \alpha$ and F logically implies $(F - \alpha \rightarrow \beta) \cup (\alpha - A) - \beta$.
    - An attribute A is extraneous in β if $A \in \beta$ and the set of functional dependencies $(F - \alpha \rightarrow \beta) \cup \alpha \rightarrow (\beta - A)$ logically implies F.
  - An algorithmic way to test whether an attribute is extraneous goes like this — for some relation schema R, set F of functional dependencies that hold on R, and some attribute A where A is involved in some dependency $\alpha \rightarrow \beta$:
    - If $A \in \beta$, look at $F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ and determine α^+ under F'. If α^+ contains A, then A is extraneous in β.

    - If $A \in \alpha$, let $\gamma = \alpha - \{A\}$ and compute γ^+ under F. If $\gamma^+ \supseteq \beta$, then A is extraneous in α.
  - Now we can get to the idea of *canonical cover for* F, written as F_c: this is the set of functional dependencies such that F logically implies all dependencies in F_c, and F_c logically implies all functional dependencies in F, and:
    - No functional dependency in F_c contains an extraneous attribute.
    - $\forall\ \alpha \rightarrow \beta\ \in\ F_c$, α is unique. Equivalently: $\alpha_1 \neq \alpha_2\ \forall\ \alpha_1 \rightarrow \beta_1, \alpha_2 \rightarrow \beta_2 \in F_c$.

- With the formalisms in place, we can now define an algorithm for computing the canonical cover:

```
Fc := F;
repeat
    replace all a1 -> b1 and a1 -> b2 in Fc with a1 -> b1 b2 (union rule);
    if a functional dependency in Fc has an extraneous attribute then
        delete the extraneous attribute from that functional dependency
    end if
until no change is made to Fc;
```

- Going back to why we're doing this in the first place, recall that maintaining integrity in a database includes testing that any changes to the data "respects" the functional dependencies defined on that data. Thus, canonical cover makes this check less expensive than the original set of functional dependencies, since it eliminates extraneous attributes and combines functional dependencies with the same left side.

- As a final footnote, F_c is not unique. You can have more than one F_c for the same F.

4 Lossless Decomposition

- With all of these normal forms and functional dependencies, you've probably noticed by now that the general tendency in normalizing a relational database is to *decompose* large relations into smaller ones, usually to decrease redundancy.

- Is it possible to check if a decomposition maintains the same data content as the original relation? Yes — that's where *lossless decomposition* comes in.

- If R is an original relation schema, F is a set of functional dependencies on R, and R_1 and R_2 are a decomposition of R. The decomposition into R_1 and R_2 is a *lossless decomposition* if, $\forall\ r(R)$ that satisfy F, $\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) = r$.

- If a decomposition isn't lossless, then it is called a *lossy decomposition*. Easy to remember, yes?

- Certain decompositions can be proven to be lossless without having to check for every possible $r(R)$ (which, generally, is impossible). For the *binary decomposition* defined above, R_1 and R_2 form a lossless decomposition if either $R_1 \cap R_2 \rightarrow R_1$ or $R_1 \cap R_2 \rightarrow R_2$, or both, are in F^+.
- In other words, if $R_1 \cap R_2$ is a superkey of either R_1 or R_2, then the decomposition of R into R_1 and R_2 is lossless.
- In practice, you use this test to assure yourself that when you're pursuing 3NF or BCNF, your decomposition of relations does not change the possible content of your data.

5 Dependency Preservation

- Another thing that we want to watch out for when decomposing relations is to make sure that we *preserve dependencies*. That is, we don't want the decomposition to result in unenforceable functional dependencies.
- We can check for this by inspection (as seen in the textbook), but as always, why do things by intuition when there is a formal or algorithmic way to do them?
- First, more definitions. For this, we're dealing with a set F of functional dependencies on a schema R, and we want to check if a decomposition $R_1, R_2, \ldots, R_n$ preserves the functional dependencies on R:
  - The *restriction* of F to R_i is the set F_i of all functional dependencies in F^+ that include *only* the attributes of R_i.
  - We care about this restriction because it is easier to check, since it involves only one relation.
  - If we define $F' = F_1 \cup F_2 \cup \cdots \cup F_n$ (the set of restrictions of F over all relations in the decomposition, a.k.a. the functional dependency sets that are easy to check because they each involve only one relation), then the decomposition is a *dependency-preserving decomposition* if $F'^+ = F^+$.
- Based on this, a generalized dependency-preservation-checking algorithm can be stated. The big bottleneck is in computing F^+ — as we've seen previously, that is expensive. We use D to represent the decomposition set $\{R_1, R_2, \ldots, R_n\}$:

```
compute Fplus;
for (schema R[i] in D) {
    F[i] := restriction of Fplus to R[i];
}
Fprime := {};
for (each restriction F[i]) {
    Fprime := Fprime union F[i];
}
comput Fprimeplus;
if (Fprimeplus = Fplus) then true else false;
```

- Alternative, less expensive algorithms are mentioned in the text, with one being much easier but only providing a sufficient and not necessary test. The other exercises the theory more, with the payoff that the algorithm can be reduced from exponential to polynomial time.

6 Decomposition via Functional Dependencies

The theoretical constructs built upon functional dependencies allow us to perform BCNF or 3NF decomposition algorithmically instead of intuitively. If you stare at the algorithms long enough, you'll see that they do pretty much match what you intuitively do when you decompose a database schema into BCNF or 3NF manually.

6.1 Decomposition to BCNF

- To decompose a relational schema into BCNF, we first have to know when that schema is *not* in BCNF. Of course the general test is to check the relation directly against the definition of BCNF, but that can be very expensive because the definition involves F^+.
- The text provides a number of simpler tests that take less work to catch a non-BCNF relation. These function as a subroutine in the actual decomposition algorithm, used as the condition for performing decomposition:

```
result := {R};
done := false;
compute Fplus;
while (not done) {
    if (there is a schema R[i] in result that is not in BCNF) {
        let alpha -> beta be a nontrivial functional dependency such that
            alpha -> beta holds on R[i],
            alpha -> R[i] is not in Fplus, // alpha is not a superkey
            alpha and beta are disjoint;

        result.remove(R[i]);
        result.add(R[i] - beta);   // yank out the dependent attributes
        result.add({alpha, beta}); // represent alpha -> beta directly
    } else {
        done := true;
    }
}
```

- The core notion of this algorithm is that you end up with a set of relations with superkeys that correspond to the left sides of some functional dependency in F^+.
- The resulting decomposition is lossless, but it is not necessarily dependency-preserving.

6.2 Decomposition to 3NF

- The decomposition algorithm to 3NF takes a different tack than the BCNF version; instead of taking a single relation then breaking it up, this algorithm starts from scratch, and actually builds the relations *from* a canonical cover of F:

```
Fc := compute a canonical cover for F;
result := {};
for (alpha -> beta: Fc) {
    if no schema in result contains {alpha, beta} {
        result.add({alpha, beta});
    }
}

if no schema in result contains a candidate key for R {
    result.add(any candidate key for R)
}
return result;
```

- Because it builds a schema from nothing, the algorithm is also known as the *3NF synthesis algorithm*.

- Note also that, because F_c is not unique for a given set F, then the algorithm can produce more than one valid 3NF decomposition, depending on which canonical cover is chosen and the order in which the functional dependencies are listed.

- This algorithm guarantees *both* a lossless and dependency-preserving decomposition. The tradeoff is that, since 3NF is not as "strict" as BCNF, there may be some redundancy of data and use of nulls.

6.3 Decomposition Bottom Line

- Through all these algorithms and definitions, this is what we want ideally from a relational database schema:

  1. BCNF
  2. lossless decomposition
  3. dependency-preserving decomposition

- However, sometimes we can't have all three, specifically BCNF *and* dependency preservation. If we hit that situation, then we choose between BCNF without dependency preservation or 3NF with dependency preservation.

- The practical upshot of all of this is that most relational database implementations base their integrity checks mainly on primary and foreign keys. Furthermore, an implementation feature called a *materialized view* allows a database to automatically perform joins on certain relations, with tests performed on the view. This frequently results in an acceptable workaround for dependency preservation while maintaining BCNF. Thus, in practice, the scale tends to tip toward choosing BCNF over dependency preservation.

7 Decomposition via Multivalued Dependencies

- But wait! There's more! Recall that, at the beginning of this whole exercise, we established that the ultimate goal of all of this is to *eliminate redundancy*.

- Unfortunately, BCNF does not completely eradicate all cases of redundancy. There isn't a precise rule that characterizes such a schema, but a general intuitive notion is for schemas that involve a relationship among entities which either have multivalued attributes or are involved in other 1-to-many relationships. Such schemas will allow copies of data without violating any functional dependencies.

- This final redundancy is dealt with using a new type of constraint, called a *multivalued dependency*, and an accompanying normal form, *fourth normal form* (4NF).

7.1 Multivalued Dependencies

- To define a multivalued dependency, we begin with the observation that a functional dependency *rules out* tuples — it prevents certain tuples from appearing in a relation. But what about the converse? What if we want to *require the presence* of a tuple? That's where multivalued dependencies come in.
- Because of this distinction, functional dependencies are sometimes referred to as *equality-generating*, and multivalued dependencies are referred to as *tuple-generating*.
- Given a relation schema R and attribute sets $\alpha \subseteq R$ and $\beta \subseteq R$. A *multivalued dependency* of β on α, written as $\alpha \twoheadrightarrow \beta$, holds on R if, for any legal relation $r(R)$, $\forall\ t_1, t_2 \in r | t_1[\alpha] = t_2[\alpha],\ \exists\ t_3, t_4 \in r$ such that:

$$t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$$

$$t_3[\beta] = t_1[\beta]$$

$$t_3[R - \beta] = t_2[R - \beta]$$

$$t_4[\beta] = t_2[\beta]$$

$$t_4[R - \beta] = t_1[R - \beta]$$

- Think about this as mix-and-match: if $\alpha \twoheadrightarrow \beta$, then two tuples with a matching α will have corresponding tuples with the same α such that the resulting four tuples match all combinations of β and $R - \beta$.
- As before, the term *trivial* is attached to the dependency if it is satisfied by all possible $r(R)$. For example, $\alpha \twoheadrightarrow \beta$ is trivial if $\beta \subseteq \alpha$ or $\beta \cup \alpha = R$ — in these cases, tuples play double duty and actually match themselves (or each other) to fulfill the above conditions, so all possible relations will indeed follow these multivalued dependencies.
- Like functional dependencies, multivalued dependencies can be inferred from each other, and so the notion of *closure* also applies: given a set D of functional and multivalued dependencies, D^+ is the set of all functional and multivalued dependencies logically implied by D.
- Also, every functional dependency is also a multivalued dependency.

7.2 Fourth Normal Form (4NF)

- With a new type of dependency comes a new normal form. The definition of *fourth normal form* (4NF) parallels the definition of BCNF; a relational schema R is in 4NF with respect to a set D of functional and multivalued dependencies if, $\forall\ \alpha \twoheadrightarrow \beta \in D^+,\ \alpha \subseteq R\ \wedge\ \beta \subseteq R$, at least one of the following is true:

  - $\alpha \twoheadrightarrow \beta$ is trivial.
  - α is a superkey of R.

- Correspondingly, a relational database schema is in 4NF if each relation schema in the database schema is in 4NF. Also, because all functional dependencies are also multivalued dependencies, then all 4NF schemas are also in BCNF.

- The concept of a *restriction* also applies to 4NF decompositions. For a set D of functional and multivalued dependencies and a decomposition $R_1, R_2, \ldots, R_n$ of some schema R, the *restriction of* D to R_i is the set D_i consisting of:

  1. All functional dependencies $\in D^+$ that include only the attributes of R_i
  2. All multivalued dependencies of the form $\alpha \twoheadrightarrow \beta \cap R_i$, where $\alpha \subseteq R_i \ \wedge \ \alpha \twoheadrightarrow \beta \in D^+$.

7.3 Decomposition to 4NF

- It turns out that the analogy between BCNF and 4NF carries over to the generalized decomposition algorithm: it is essentially the same, except that it uses multivalued dependencies and the restriction of D^+ to R_i. The full algorithm is in the text.

- Again analogous to BCNF, the resulting 4NF decomposition can be shown to be lossless. The analogous definition of a lossless decomposition with respect to multivalued dependencies uses D^+ instead of F^+, requiring that at least one of these multivalued dependencies is in D^+:

  - $R_1 \cap R_2 \twoheadrightarrow R_1$
  - $R_1 \cap R_2 \twoheadrightarrow R_2$

8 And So Forth

- Yes, there are more types of dependencies, and yes, there are more types of normal forms that go with them. We'll just mention them here, and you can get to these if you go to graduate school.

- *Join dependencies* generalize multivalued dependencies, and satisfying those leads to the *project-join normal form* (PJNF), which some call *fifth normal form* (5NF).

- There are even more general constraints that lead to another normal form called *domain-key normal form* (DKNF).

- As the dependencies required by these normal forms get more and more general, the theory and reasoning on these constraints gets harder and harder. Thus, at the moment, these normal forms tend to live primarily in the world of theory, and have not seen widespread practical use.

- But it's good to at least know about them, and know about how they fit in the big picture.

- And as always, the bottom line of all this is to: (1) fully understand the rules that govern your database (based on the application domain) by formally defining them, and (2) creating a database design that minimizes redundancy and maximizes automation — let the database be the one to enforce the integrity of your data, so that your application code doesn't have to worry about it.